Rapport de fin d'étude deuxième cycle Analyse statique pour l'optimisation de spécification SDL.

Cyril Alexandre Pachon

Date : 31 Aout 2000

Laboratoire de recherche VERIMAG
2, av. de Vignate 38610 Gières.

Table des matières

Table des figures

1 Introduction.

Ce document est un rapport d'activité sur le stage de dix semaines effectué en fin de MST ESI 2. Il expose les différentes phases de mon stage et les résultats obtenus.

<u>Il est constitué :</u>

- D'une brève présentation du laboratoire Vérimag et de ses activités,
- Du domaine dans lequel se situe mon travail pour effectuer ce stage,
- De mon travail :
  - Les objectifs;
  - Les méthodes et les outils utilisés;
  - La réalisation donnant lieu à l'état d'avancement.
- De mon bilan sur ce stage.

2 Contexte.

2.1 Le laboratoire Vérimag.

Mon stage se déroule au sein du laboratoire de recherche VERIMAG 2, av. de Vignate 38610 Gières.

Vérimag est une unité mixte de recherche du CNRS, de l'INPG et de l'UJF, et fait partie de la fédération IMAG. Vérimag est en fait la continuation de l'unité industrielle Vérimag, créée en Janvier 1993 par la société Vérilog et les autorités de tutelle de l'Institut IMAG. Elle vise à produire des méthodes et outils pour le développement de systèmes critiques réactifs.

Les travaux menés à Vérimag visent à développer des techniques de validation pour deux familles de langages utilisés pour la description des applications critiques. Les langages synchrones comme les langages "Lustre" et "Argos" développés à Vérimag, dédiés à la programmation des systèmes de commande temps réel. Les langages asynchrones pour la spécification des systèmes distribués et notamment des protocoles tels que les langages "Lotos" et "SDL" (qui sont des normes internationales) pour lesquels des environnements de validation dédiés sont mis en oeuvre. Vérimag joue un rôle important dans le domaine de l'application des méthodes formelles au développement des systèmes critiques. Il réunit des compétences en programmation synchrone, spécification des protocoles et des systèmes distribués, analyse des systèmes temporisés et hybrides.

2.2 Validation de systèmes distribués.

Au sein de Vérimag, mon stage se situe dans le cadre de la validation des systèmes distribués, et, plus précisément autour de l'environnement de validation pour des programmes "SDL" qui est développé dans ce laboratoire. L'architecture globale de l'environnement est présentée dans la figure 1. Cet environnement est centré autour de la représentation intermédiaire *IF* (*Intermediate Form*, développé et expliqué en 3.1). Dans ce contexte *IF* assure le lien entre des formalismes de description de haut-niveau, notamment des spécifications "SDL", et des modèles sémantiques de bas niveaux, notamment les automates temporisés et les systèmes de transitions. C'est au niveau des optimisations sur *IF* que nous allons pouvoir réduire le système de transitions. Nous pourrons agir pour en limiter les redondances au niveau des variables, mais aussi limiter l'explosion des états. Le travail devient donc concentrique sur le module *IF*. A cette étape nous avons l'outil *LIVE* (voir section 3.2) qui implémente l'analyse des variables actives et les optimisations syntaxiques induites sur les programmes *IF*. Il comporte une composante pour réaliser l'abstraction des variables. Intuitivement, il prend en entrée un programme *IF* et fournit en sortie un autre où toutes les redondances produites par les variables inactives ou inutiles sont éliminées explicitement par l'introduction systématique de réinitialisations (voir section 3.2.2). C'est donc dans ce cadre que je dois exploiter l'outil *LIVE* pour supprimer les actions redondantes, ainsi que certaines variables inutiles. De ce fait au lieu de rajouter des actions notifiant la suppression, je les supprimerai en respectant la cohérence du résultat final, c'est à dire du système de transitions.

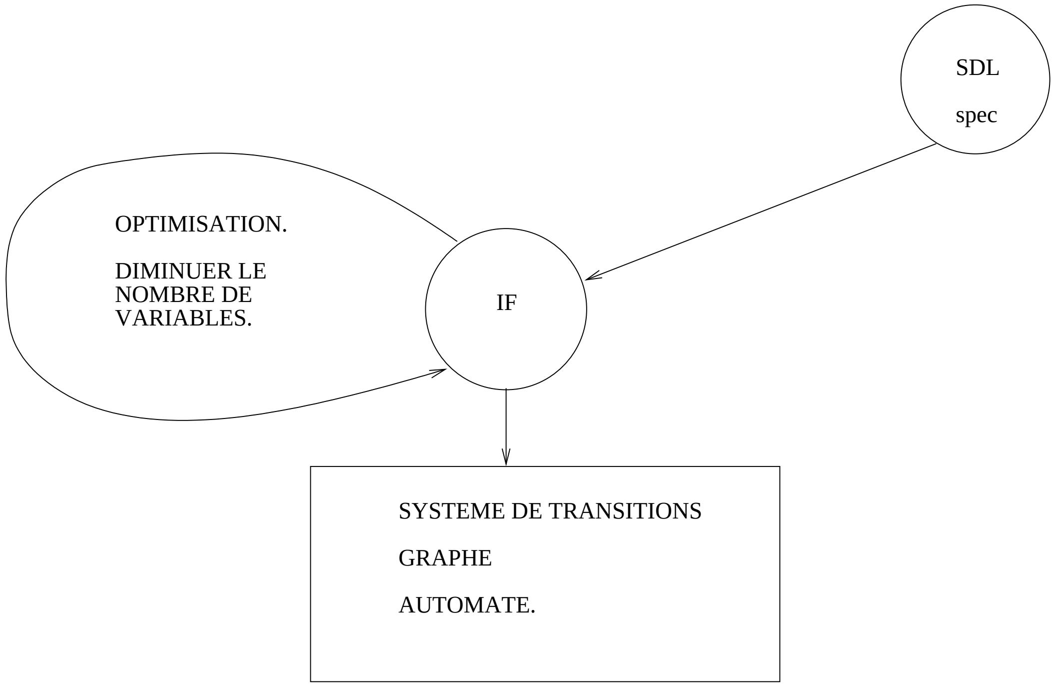


FIG. 1 – *Ensemble des séquences d'exécution.*

2.3 Sujet de mon stage.

Contexte : Optimisation d'un programme *IF* (*Intermediate Form*).

Sujet : Mettre en oeuvre une technique de recouvrement de variables, basée sur le nombre maximal de variables simultanément actives (une variable est active dans un état, si elle est utilisée avant d'être redéfinie dans une action de cet état) . En effet, le nombre de variables nécessaires à l'exécution du programme correspond exactement au nombre de variables simultanément actives, les variables supplémentaires pouvant être supprimées. Je peux donc établir la formule suivante qui sera développée par la suite.

$\forall$ l'état,$\sum$ (Variables nécessaires) = Max ($\sum$ (Variables actives))

Si $\sum$ (Variables du programme) $>$ $\sum$ (Variables nécessaires) $\Longrightarrow$ Un recouvrement est possible.

Le recouvrement consiste à diminuer le nombre de variables. Pour cela, il suffit de ne garder que le nombre maximum de variables nécessaires de chaque état. Ce nombre, plus petit que le nombre de variables initiales du programme, donnera la quantité des nouvelles variables de "renommage".

Plus précisément, l'objectif de mon stage est de réaliser un outil capable de décider si un programme *IF* contient ou non des variables redondantes et, le cas échéant, de générer un programme *IF* équivalent contenant exactement le nombre de variables nécessaires. L'intêret est alors de diminuer le nombre de variables du programme, ce qui peut réduire de manière spectaculaire le nombre des états de l'automate généré lors de la spécification. Cet outil s'appuie sur des librairies et composants déjà réalisés à Vérimag, et il doit s'intégrer dans l'environnement *IF* dédié à la validation de spécification "SDL".

3 Les outils disponibles.

Dans un premier temps, cette partie sera constituée de la présentation du langage *IF* (en 3.1). Je regarderai comment un programme *IF* se réalise. J'examinerai plus en détail, ses déclarations, et les différentes actions que l'on peut établir. Puis j'illustrerai ma description de ce langage par un exemple complet noté *essai* (en 3.1.3). Ensuite cet exemple sera traduit en automate (en 3.1.4).

Dans un deuxième temps, je décrirai l'outil *LIVE* (en 3.2), qui est à ma disposition, et qui me servira pour la réalisation du travail demandé.

J'étudierai à partir d'un exemple donné l'explication de la génération d'un programme *IF* par *LIVE* pour montrer les différentes informations que *LIVE* peut nous fournir (3.2.1). Puis je donnerai une partie du nouveau programme *IF* obtenu (en 3.2.2).

3.1 Le langage *IF*.

A ce niveau je vais vous présenter le langage *IF* utilisé puis, j'argumenterai cette présentation par un exemple de programme (en 3.1.3).

3.1.1 Syntaxe du langage *IF*.

Pour débuter le stage il fallait d'abord comprendre comment un protocole "SDL" est traduit en *IF* (*Intermediate Form*). Car, pour établir une jonction entre les langages de haut-niveau comme "SDL" et en faciliter la vérification, Vérimag a développé un langage plus simple, le format *IF*. J'ai donc dû effectuer mes premiers programmes en *IF* afin de me familiariser avec ce langage.

Un système *IF* est constitué d'un ensemble d'automates temporisés, qui communiquent soit de manière asynchrone par envoi de signaux à travers des files d'attente, soit de manière synchrone par rendez-vous aux portes de synchronisation. Plus généralement ce format peut-être vu comme un "assembleur" pour les langages de spécification comme "SDL". Sans tenir compte pour l'instant du fait qu'un protocole peut avoir des synchronisations entre différents processus, nous allons voir syntaxiquement ce que contient un programme *IF* :

Aspect général :

C'est un ensemble de processus, donc chacun est nommé et comporte :

des types de donnés;
des signaux de communication asynchrone;
des portes des synchronisations;
des files d'attente (de type FIFO);
des variables partagées (ou globales);
des états (source ou cible);
des transitions;
des actions traduites en expressions;
et une expression de synchronisation entre processus.

Nous allons voir les différents mots clefs du langage.

```
system:=
  SYSTEM system_id;  /* Nom du système. */

  TYPE          /* Déclaration des types. */
  SIGNAL        /* Déclaration des signaux. */
  GATE          /* Déclaration des portes. */
  BUFFER        /* Déclaration des files d'attente. */
  VAR           /* Déclaration des variables. */
  SYNC sync-expression END;  /* Synchronisation des expressions.*/

  PROCESS...     /* Corps des processus. */
```

Après avoir listé les mots clefs, voyons comment nous les utilisons.

Les signaux sont les objets transportés de manière asynchrone.
Un signal est représenté par un NOM DE LISTE et son TYPE.

signal :
$signal - id[(type - id, type - id*)];$

Les files d'attente (ou buffers) assurent la réalisation de la communication asynchrone dans un système.

Elles sont identifiées par un NOM unique et elles transportent des signaux envoyés entre états, ou entre processus et l'environnement du système. Il n'y a pas de restrictions d'utilisation de files. Chaque processus peut envoyer des signaux à travers toutes les files du système.

buffer :
buffer-id:buffer-class OF signal-id1,..signal-idn;

Les portes servent à réaliser la communication synchrone dans un système *IF*. Ainsi, deux ou plusieurs processus peuvent se synchroniser et échanger des valeurs par l'intermédiaire d'une porte. Chaque porte précise la liste des types de valeurs qui peuvent y être échangés.

gate :
gate-id(type-id1,..type-idn);

L'expression de synchronisation décrit comment les processus communiquent de manière synchrone les uns avec les autres, en utilisant l'opérateur binaire de composition parallèle avec synchronisation sur un ensemble de portes.

sync-expression :
process-id
| sync-expression | [gate-id1,.. gate-idn] | sync-expression

Un processus se décompose alors par ses variables locales; ses états; ses transitions..... Un processus est représenté par son NOM

PROCESS {nom};

Les états sont des éléments importants dans la description du contrôle de processus. Les états correspondent au mode de fonctionnement des processus : à tout moment un processus est dans un état et suivant les évènements internes ou externes qui se produisent il peut passer dans un autre. Un état peut être initial, il sera un point d'entrée pour le début des communications. Un état peut être stable ou instable. Cette propriété concerne l'atomicité des séquences d'exécution. Un processus ne pourra pas être interrompu s'il est dans l'état instable. Il est représenté par un NOM unique, si c'est un état de transition. Ou il est identifié par un NOM et INIT , s'il s'agit de l'état de départ.

state :
state-id : INIT;
|
state-id;

Tous les états évoluent en exécutant des transitions. Une transition peut-être synchrone ou asynchrone. Une transition décrit l'opération permettant à un processus de passer d'un état à un autre.
Elle est décomposée de la façon suivante :

Au niveau des états, une transition comporte toujours un état source introduit par FROM et d'un état cible précédé de TO. A l'intérieur est décrite l'action à faire pour passer d'un état à l'autre.

transition :
FROM state-id
transition-body
$TOstate - id$;

Le corps d'une transition est constitué de différentes actions. Les actions élémentaires associées aux transitions sont des réceptions ou émissions asynchrones, des synchronisations ou des simples affectations.

transition body :
IF expression
[input] output1,.. outputn
assign1,..assignn
|
IF expression
synchronous-input
assign1,.. assignn

Regardons la composition des actions de communication :

input :
INPUT signal-id(variable-id1,..variable-idn)
FROM buffer-id
IF expression

output :
OUTPUT signal-id(expression-id1,.. expression-idn)
TO buffer—expression

synchronous-input :
INPUT gate-id offer1,.. offern
IF expression

Puis nous pouvons aussi avoir des conditions.

offer :
?variable-id—!expression

Nous pouvons trouver des affectations simples comme le ":=" qui affecte à la partie gauche la valeur de la partie droite (cf le TASK). Mais nous avons aussi des affectations plus spécifiques comme l'indication d'une suppression d'une variable (cf le RESET), ou simplement la récupération de la variable (cf le SET).

assign :
SET variable-id
|
RESET variable-id
|
TASK variable-id:=expression

Une expression est constituée soit d'une variable, soit d'une constante. Mais aussi elle peut être décrite par une structure ou une fonction.

Expression :
variable-id—function-id(expression(1), .. ,expression(n))

Les données peuvent être locales, ou globales toujours précédées de la notification VAR.

Une variable est identifiée par un NOM UNIQUE et est typée. A ce niveau elle est locale, donc définie et visible à l'intérieur d'un processus *IF*.

variable :
variable-id: type-id;

Pour tout programme *IF*, on peut associer un système de transitions étiquetées qui représente un ensemble de ses séquences d'exécutions.

3.1.2 Conception d'un programme en *IF*.

Tout d'abord un programme commence par le label *system* suivit d'un NOM.

```
system NOM;
```

Ensuite on déclare successivement l'environement du *PROCESS*, *les signaux, les files d'attente, le nom du processus, et les variables globales qui composent le programme.*

```
signal
sign (int);
```

```
buffer
file : queue of sign
```

```
process ESSAI;
```

```
var
X : INT;
Y : INT;
```

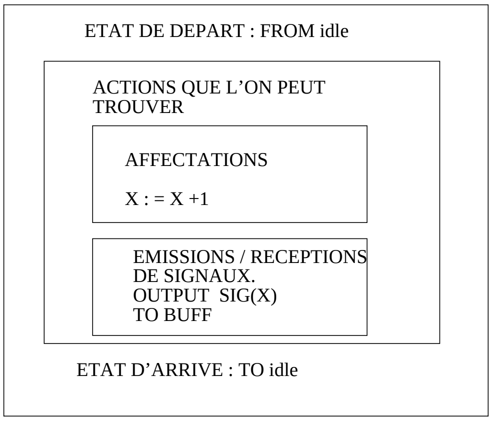
Les états correspondent au mode de fonctionnement des processus. Un programme *IF* doit comporter un état initial, il sera noté INIT. Puis on explicite les transitions.

```
STATE
```

```
Q0: INIT;
```

```
TRANSITION
```

Puis enfin les transitions entre chaque état comportent les actions à effectuer (nous supposons, pour le travail demandé ici, que chaque transition ne comporte qu'une seule action).



3.1.3 Exemple d'un programme *IF* noté *essai*.

Cet exemple sera constitué d'un signal noté *SIGN*, d'une file notée *FILE*, d'un seul process *ESSAI*, de six variables locales *a, b, c, d, e, f*, et de sept états *q0, q1, q2, q3, q4, q5, q6*.

```
system essai;

signal
  sign(int);

buffer
  file: queue of sign;

/*——————————————————————————*/

process essai;

var a: int;     /* Déclaration des variables.*/
  b: int;
  c: int;
  d: int;
  e: int;
  f: int;

state           /*  Déclaration des états.    */
q0:init;
q1;
```

```
q2;
q3;
q4;
q5;
q6;

transition      /*  Liste des transitions.  */

from q0
do
  a := e
to q1;

from q1
do
  b := b + 1
to q2;

from q2
do
  a := b + c
to q2;

from q2
do
  output sign(a) to file
to q2;

from q2
do
  d := d * 1
to q3;

from q2
do
  c:= d * a * c
to q4;

from q2
do
  b := a - 6
to q5;

from q2
if (a > 1)
```

```
do
  a := 1
to q6;

from q6
do
  output sign(b) to file
to q1;
```

3.1.4 Traduction du progamme *IF essai* en automate.

Les cercles représentent les différents états présentés dans le "process essai", **q0** étant l'état initial. Les flèches sont les transitions de communication entre les états. Et les actions sont écrites sur chaque transition.

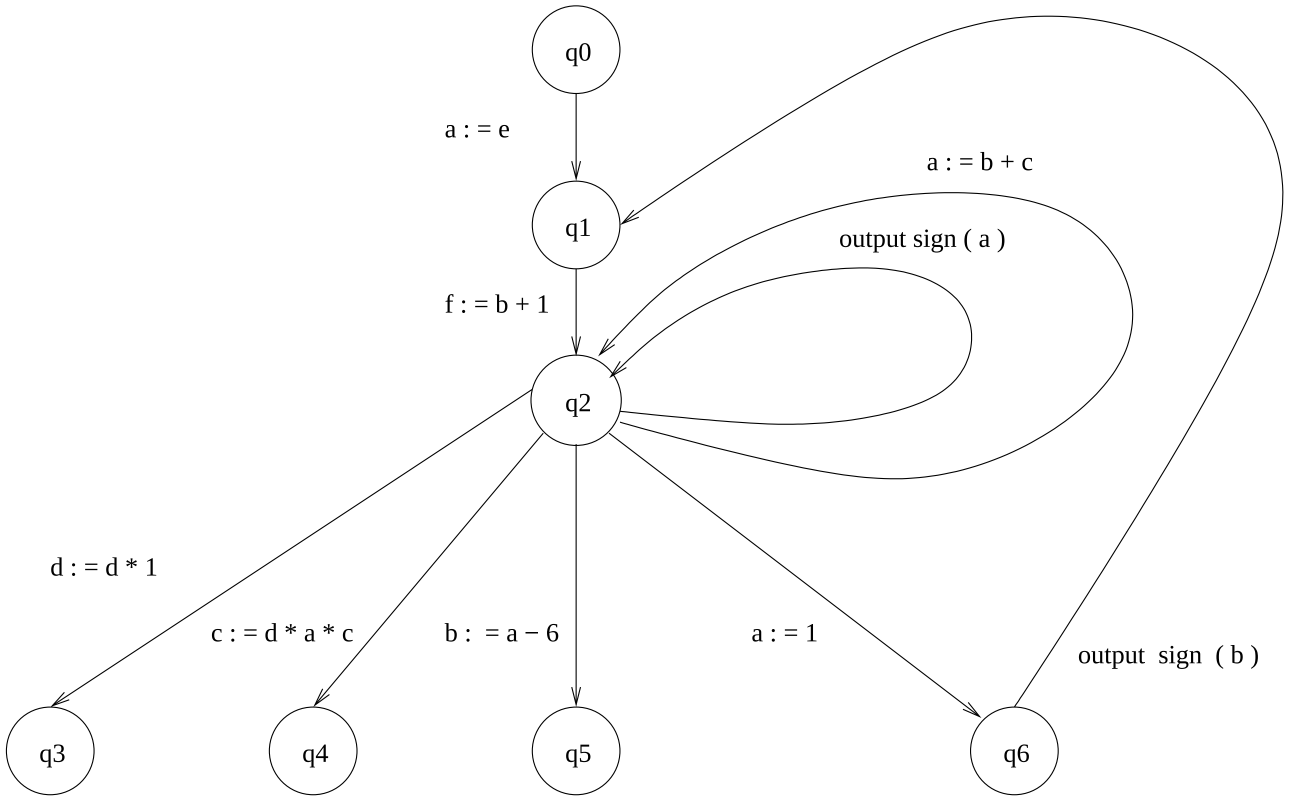


Fig. 2 – *Automate du programme IF essai présenté en 3.1.3.*

3.2 L'outil *LIVE*.

Pour me permettre de réaliser le travail demandé, j'ai dû étudier le programme *LIVE*. Il consiste, à partir d'un programme *IF*, à connaître les variables actives au niveau de chaque état. J'étudierai *LIVE* à partir de l'exemple *essai* présenté en 3.1.3. Ceci va nous permettre de voir ce que génère *LIVE*. Je vais d'abord mettre le résultat de la première phase du travail du programme, que je découperai pas à pas pour vous donner ses principales fonctionnalités. Puis je donnerai le nouveau programme *IF* qui sera notre nouveau modèle.

3.2.1 Analyse de la première phase de l'exécution *LIVE* sur *essai*.

/*Live analysis for SYSTEM essai

PROCESS essai

VARS
1 - a
2 - b
3 - c
4 - d
5 - e
6 - f

$DEAD_vars = \{f\}$

STATE q0
$(LIVE_vars = \{2, 3, 4, 5, \})$

STATE q1 et même résultat pour q2
$(LIVE_vars = \{1, 2, 3, 4, \})$

STATE q3 et même résultat pour q4 et q5
$(LIVE_vars = \{\})$

STATE q6
$(LIVE_vars = \{1, 2, 3, 4, \})$

TRANSITION from q0 do { a:=e } to q1;
$(DEF = \{1,\}; USE = \{5,\}; AllUses = \{5,\})$

TRANSITION from q1 do { f:=(b + 1) } to q2;
$(DEF = \{6,\}; USE = \{2,\}; AllUses = \{2,\})$

TRANSITION from q2 do { a:=(b + c) } to q2;
$(DEF = \{1,\}; USE = \{2,3,\}; AllUses = \{2,3,\})$

TRANSITION from q2 do { output sign(a) to file } to q2;
$(DEF = \{\}; USE = \{1,\}; AllUses = \{1,\})$

TRANSITION from q2 do { c:=((d * a) * c) } to q4;
$(DEF = \{3,\}; USE = \{1,3,4,\}; AllUses = \{1,3,4,\})$

Je expliquerai plus en détail ce que signifie cette exécution :

PROCESS ESSAI;
VAR
1 - A;
2 - B;
3 - .....
$DEAD_vars = \{ \text{F} \}$

Tout d'abord nous avons la partie déclarative. Le complément DEAD_vars nous indique que les variables de cet ensemble ne sont jamais utilisées. Ces variables peuvent donc être supprimées. Par exemple ici la variable **f** est supprimée. Une anticipation serait que toutes les actions utilisant les variables dans DEAD seront aussi supprimées (Nous allons pouvoir le constater dans le programme *IF* que redonne *LIVE* par la suite).

STATE q0
$(LIVE\_vars = \{2,3,4,5,\})$

STATE q1 et même résultat pour q2
$(LIVE\_vars = \{1,2,3,4,\})$

STATE q3 et même résultat pour q4 et q5
$(LIVE\_vars = \{\})$

Cette partie nous donne l'information sur l'activité des variables. On peut noter que dans l'état q0 quatre variables sont actives **b, c, d, e**. Et que dans l'état *q3, q4, q5* aucune des variables n'est active. Ce qui nous conforte dans le fait que **f** soit DEAD c'est qu'elle n'apparaît pas dans les différents ensembles $LIVE\_vars$.

TRANSITION from q2
do {
 a:=(b + c)
}
to q2;
$(DEF = \{1, \}; USE = \{2, 3, \}; AllUses = \{2, 3, \})$

Ici nous avons des informations plus précises sur les variables à l'intérieur même des actions. En effet le DEF nous renseigne sur la redéfinition de la variable (variable située en partie gauche d'une affectation). Le USE nous dit si une variable est utilisée en partie droite de l'expression. Si nous prenons une action In/Out la variable est USE.

L'ensemble ALLUSES comporte les variables à la fois DEF et USE ou uniquement USE. Si une variable est DEF dans une action, (c'est à dire qu'elle se retrouve en partie gauche d'une expression), puis que dans l'affectation suivante on la réutilise en partie droite, alors sa référence est visible dans l'ensemble DEF et non dans l'ensemble USE. C'est donc pour garder l'information sur le USE de cette variable qu'elle apparaîtra dans l'ensemble ALLUSES. Ceci est le fait d'avoir plusieurs actions pour une transition, nous n'avons pas d'exemple dans ce document.

Un exemple peut être donné à la main :

(DEF = {3, 4, }; USE = {1, 3, }; AllUses = {1, 3, 4, })

Ceci nous indique que la variable **4** est aussi USE grâce au ALLUSES.

3.2.2 Résultat du modèle *IF*, avec le programme *LIVE* initial.

En partant du programme *essai* (en 3.1.3), le programme *LIVE* détermine les utilités de chaque variable, rajoute des *RESET* pour toute variable inactive pour la suite du programme, et supprime les variables et les actions des variables mortes (variables contenues dans l'ensemble DEAD_vars du départ). Je présente ici les informations principales traduites par *LIVE*, le reste du programme sera en annexe.

Le début reste le même qu'avant.

var

a : int;
b : int;
c : int;
d : int;
e : int;

ici il manque la variable f

state
{ **IDEM** }

transition

from q0
do {
a:=e,

reset e A partir de là, la variable e n'étant plus utilisée pour la suite du programme, elle est **RESET**. C'est donc le *LIVE* qui rajoute l'action *reset variable.*

}
to q1;
from q1

Nous constatons ici que l'action f:=(b + 1) est supprimée du fait que la variable f est une variable DEAD.

to q2;

from q2
do {
a:=(b + c)

Ici a n'est pas reset. On constate que nous avons besoin de a pour cette action, mais aussi pour l'autre communication d'après qui est *output sign(a) to file*, **donc il ne faut pas la supprimer.**

}
to q2;

from q2
do {
output sign(a) to file
}
to q2;

```
from q2
  do {
   d:=(d * 1),
   reset a,     A ce niveau plus d'utilisation des variables a, b, c, d.
   reset b,
   reset c,
   reset d
  }
  to q3;
```

4 Réalisation.

Cette partie sera la présentation de mon travail. Je présenterai d'abord un exemple de programme *IF* sous la forme d'un automate. Le code de ce programme est présent dans l'annexe, il s'agit du programme *échange* . Puis je vais donner ce que l'on veut réellement obtenir (toujours sous la forme d'un automate). Enfin je donnerai l'algorithme formel sous forme de quatre étapes, constituant l'exécution du programme de recouvrement.

4.1 Un exemple d'introduction.

4.1.1 Automate initial étudié.

Pour voir toutes les difficultés, nous allons prendre un exemple qui contient un cycle. Nous prenons comme état de départ **q0** et nous bouclons sur lui par **q6** . Chaque transition comporte une seule action, qui est notée sur la transition. Le **OUTPUT** correspond à une communication simple entre deux états comme un IN ou un OUT. A chaque état, nous donnons sous forme de liste, les variables qui sont actives.

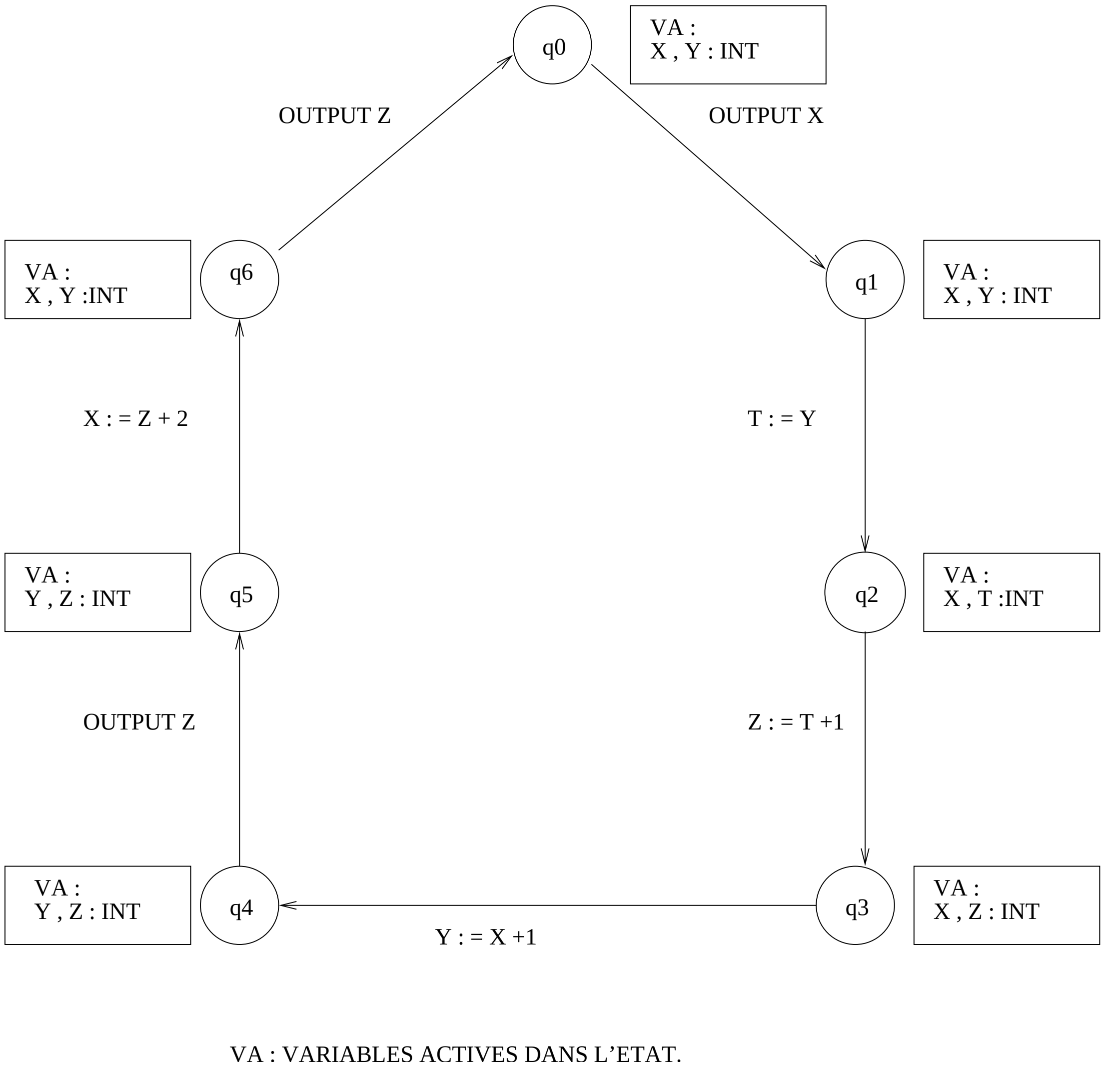


FIG. 3 – *Visualisation par états du nombre de variables, avec l'action sur la transition.*

4.1.2 Automate final que l'on souhaite obtenir.

Le programme précédent comporte quatre variables (x, y, z et t). Toutefois en regardant la *figure 3* (en 4.1.1), nous pouvons constater que les états ne comportent à chaque fois que deux variables actives. C'est pour cela que nous ne pouvons réécrire le programme en utilisant que deux nouvelles variables **V0 et V1**. Mais attention à garder la cohérence du résultat. C'est pourquoi les nouvelles actions (affectations) comportent **Tmp** entre **q6 et q0** (en 4.2.2). Cette introduction de la variable **Tmp** sera expliquée par la suite (en 4.2.3).

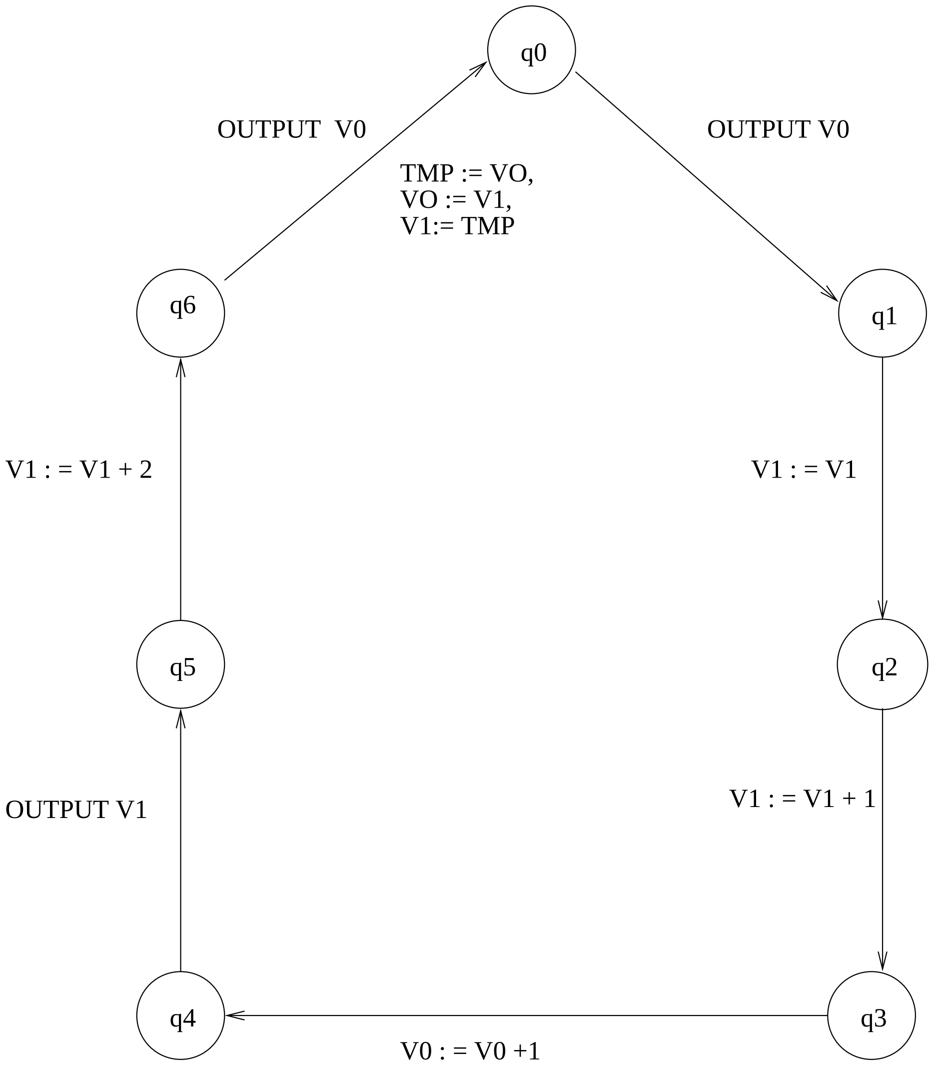


FIG. 4 – *Visualisation de l'automate de départ après réduction du nombre de variables.*

4.2 Mise en oeuvre.

Dans cette partie je décrirai l'algorithme informel pour effectuer le "renommage" des variables dans un programme *IF*. Dans le même temps je pourrai supprimer les actions redondantes (inutiles).

Cet algorithme est constitué de **quatre étapes** dont les deux premières (en 4.2.1 et 4.2.2) sont là mise en place du "renommage". Elles sont la pour envisager le "renommage" (constituant l'étape 1). Si le "renommage" est décidé selon les critères imposés, elles incluent la phase d'initialisation des structures de données, qui se trouvent être ici un grand tableau composé de plusieurs champs (constituant l'étape 2) . Puis nous avons la phase de suppression des actions redondantes (en 4.2.3, constituant l'étape 3). Enfin la dernière phase est une lecture du tableau pour reproduire le modèle final, qui est un programme *IF* plus petit que l'initial mais de cohérence identique. C'est à ce stade que, selon les informations précédentes, nous ferons nos échanges de variables comme décrites en 4.1.2 (constituant l'étape 4).

<u>Le programme se présente de la façon suivante :</u>

bien sûr les références **TR[].WR, Mq, See, exchange** seront expliquées (en 4.2.2).

/* Étape 1. */

Pour tous les états du programme *IF*:
Récupération du nombre de variables nécessaires.

Si nombre de variables nécessaires < nombre de variables du programme **alors**

/* Étape 2. */

Pour tous les états du programme *IF*:
On regarde quelles sont les variables actives, et on met à jour, avec la valeur 0, le champ TR[].WR initialement égal à -1.

Pour tous les états du programme *IF*:
On met à jour dans le tableau le champ Mq (celui du "renommage"), en prenant soin de prendre des marques différentes pour chaque variable active (TR[]WR = 0).
On met à jour le champ See (indique que la variable de l'état est déjà vue, See = 1).

/* Étape 3. */

Pour tous les états du programme *IF*:
On met à jour le champ de l'échange de deux variables (exchange).
On supprime les actions dont les variables ne sont plus dans l'ensemble de celles actives dans l'état cible.

/* Étape 4. */

Pour tous les états du programme *IF*:
On lit le tableau et on renomme.

Sinon on continue le programme *LIVE*.

On affiche le nouveau programme *IF*.

4.2.1 Peut-on renommer ?

Dans un premier temps je détermine si nous avons la possibilité de renommer, ceci étant réalisé en modifiant le programme *LIVE* existant. C'est à dire à partir des variables actives de chaque état, je regarde si je peux faire un gain de variables. En regardant la *figure 3* (en 4.1.1), on s'aperçoit très vite que seules deux variables sont utilisées en même temps dans un état. Donc on peut réduire le nombre de variables.

<u>On applique à ce stade :</u>

Le critère de "renommage" est basé sur la formule suivante :

$\forall$ l'état, $\sum$ (Variables nécessaires) = Max ($\sum$ (Variables actives))
Si $\sum$ (Variables du programme) > $\sum$ (Variables nécessaires) $\Longrightarrow$ Un recouvrement est possible.

Si la réponse est non, le programme *LIVE* termine normalement.

Si la réponse est oui, nous commençons le programme pour renommer en récupérant le nombre de variables nécessaires au "renommage". Si nous faisons référence à la *figure 3* (en 4.1.1), nous avons l'impression que nous pouvons renommer le programme uniquement avec deux variables au lieu de quatre. Nous allons voir ce qu'il en est.

4.2.2 Générer le "renommage" des variables.

On fait une première passe sur le programme pour déterminer les variables à renommer. La structure de données sera un grand tableau. Nous nous fixons un point de départ, par exemple l'état INIT **q0**. Nous commençons par marquer les variables à renommer. On se base sur le module *LIVE* qui nous indique quelles sont les variables actives au niveau de l'état. La marque à ce niveau sera un 0 si la variable est à renommer et -1 sinon (correspondant au champ TR[].WR).

<u>Exemple de résultat du tableau pour l'état initial de la *figure 3*:</u>

V:x TR[].WR:0 **Src:q0**	Src indique la source de l'action.
V:y **TR[].WR:0** Src:q0	Le TR[].WR est la marque de la variable active.
V:z TR[].WR:-1 Src:q0	V est le nom de la variable dans l'état.
V:t TR[].WR:-1 Src:q0	

Pour le reste de la construction du tableau, c'est à dire pour les autres états, elle se fait en relation avec le résultat de l'état source précédent. De ce fait nous gardons au maximum le même nouveau nom pour une variable (ceci est traduit par la valeur du champ **Mq**). Si le "renommage" de la variable doit changer, il prend un nom qui n'est pas encore utilisé mais utilisable (le but étant de garder la cohérence du résultat). C'est pour cela que j'ai introduit une marque de passage de l'état pour ainsi vérifier la continuité du "renommage" et éviter d'écraser des valeurs indispensables. Enfin le champ **See** initialisé à 0 indique si la variable de l'état a déjà été vue. Dans ce cas on marque le champ par 1.

<u>Visualisation au niveau du tableau pour l'exemple de la *figure 3*:</u>

V:x TR[].WR:-1 Src:q4 Mq:-1 See:0
V:y TR[].WR:0 Src:q4 **Mq:0 See:1** la marque est 0 et 1 pour y , z
V:z TR[].WR:0 Src:q4 **Mq:1 See:1** état vu pour See = 1 pour cette variable.
V:t TR[].WR:-1 Src:q4 Mq:-1 See:0

V:x TR[].WR:-1 Src:q5 Mq:-1 See:0
V:y TR[].WR:0 Src:q5 Mq:0 See:1 On garde le même "renommage" le plus longtemps possible: tant que nous avons l'information que la variable reste active, tant que nous avons assez de variables pour effectuer notre "renommage". Nous gardons les noms et utilisons les noms disponibles *voir figure 5* (en 4.2.4).

V:z TR[].WR:0 Src:q5 Mq:1 See:1
V:t TR[].WR:-1 Src:q5 Mq:-1 See:0

Ainsi de suite jusqu'à la fin du nombre d'états. Et là on s'aperçoit déjà que nous devons procéder éventuellement à des échanges entre les nouveaux noms et les variables existantes. Plus précisément : si vous êtes dans le cas où la variable X est renommée en V0 et la variable y en V1 dans l'état source, ici **q6** *figure 4* (en 4.1.2), et que l'on doit utiliser X en V1 et y en V0 dans l'état cible **q0** de la *figure 4* (en 4.1.2), il faut recréer la correspondance. Pour cela nous devons effectuer des échanges de variables, et introduire de ce fait une variable supplémentaire que l'on appellera **Tmp**.

Soit pour l'exemple, il faut intervertir les valeurs de **V0 et V1** qui sont en réalité X et Y. Cette échange est indispensable pour garantir la cohérence et le même résultat que pour le modèle initial.

Tmp := V0,
V0 : = V1,
V1 : = Tmp

Nous sommes obligés de faire cet échange sinon notre modèle final sera faux. Donc pour le test du "renommage" il faut ajouter une variable de plus, nous avons **V0, V1, et Tmp**.

Soit la nouvelle formule à considérer :

Si $\sum$ (Variables du programme) > $\sum$ (Variables nécessaires) + 1 $\Longrightarrow$ Un recouvrement est possible.

4.2.3 Supprimer les actions redondantes.

Pour considérer une action redondante, il faut regarder si nous utilisons la variable dans la suite du programme.

Par exemple :

Si dans une action on effectue X : = 3,
et que dans l'état d'après nous faisons X := 4,

alors, s'il n'y a pas d'utilisation de la variable X entre les deux affectations, la première action devient une action redondante que l'on peut supprimer. Le principe est qu' une variable inactive dans l'état cible peut être supprimée, de même que l'action qui la comporte.

4.2.4 Appliquer le "renommage".

Après avoir rempli la structure de données, nous pouvons l'exploiter pour faire le "renommage". Nous avons à ce stade toutes les informations dans le tableau et de plus nous savons si, lors d'une boucle, nous avons des variables à échanger (voir 4.2.2 pour déterminer l'échange de variables).

Rappel de la structure complète avec référence de l'échange:

référence de l'échange :

La structure du tableau comporte une information sur les éventuels échanges.

tabRenown[0] = X tabRenown[4] = X

Cette information nous guidera pour les éventuels échanges (0 et 4 sont les indices des variables (ici X) dans le tableau de "renommage"). En effet, dans cet exemple nous avons une boucle sur plusieurs états et nous avons dû renommer la variable X (c'est le champ tabRenown[] qui nous indique un échange possible). Pour des problèmes de cohérence ou de non disponibilité du même nom de "renommage", la variable X peut avoir deux noms différents. C'est pour cela qu'il faut le noter en vu de rétablir la cohérence de la variable en entrée et en sortie de l'état (dans l'exemple *échange* disponible en annexe nous pouvons constater qu'au passage de l'état q6 à q0 nous avons un échange. Ceci est visible aussi sur l'automate *figure 4* (en 4.1.2)).

Structure complète :

V:x TR[].WR:0 Src:q0 Mq:0 See:1 Le renom des variables sera
V:y TR[].WR:0 Src:q0 Mq:1 See:1 x en V0 et y en V1.
V:z TR[].WR:-1 ...
V:t TR[].WR:-1 ...

V:x TR[].WR:0 Src:q6 Mq:1 See:1 Le renom des variables sera
V:y TR[].WR:0 Src:q6 Mq:0 See:1 x en V1 et y en V0
V:z TR[].WR:-1 ...
V:t TR[].WR:-1 ...

L'information de "renommage" pour une action se situe au niveau de son état source et de son état cible.

Prenons le cas d'une affectation. Elle peut se décomposer en deux parties : une partie gauche du signe d'affectation " := " et une partie droite.

Pour transmettre les informations et garder la cohérence dans l'affectation, nous allons procéder au renom de la façon suivante :

TRANSITION from q1 *LIVE* au niveau de q1 ($LIVE_vars = 1, 2,$)
do
t:=y
to q2;
LIVE au niveau de q2 ($LIVE_vars = 1, 4,$)

TRANSITION from q2
do
z:=(t + 1)
to q3; *LIVE* au niveau de q3 ($LIVE_vars = 1, 3,$)

TRANSITION from q3
do
y:=(x + 1)
to q4; *LIVE* au niveau de q4 ($LIVE_vars = 2, 3,$)

Information de renom au niveau q1 :
V:x Src:q1 Mq:0 x en V0
V:y Src:q1 Mq:1 y en V1

Information de renom au niveau q2 :
V:x Src:q2 Mq:0 x en V0

V:t Src:q2 Mq:1   t en V1

Information de renom au niveau q3 :
V:x Src:q3 Mq:0   x en V0
V:z Src:q3 Mq:1   z en V1

Information de renom au niveau q4 :
V:y Src:q4 Mq:0   y en V0
V:z Src:q4 Mq:1   z en V1

<u>Nous donnant ainsi :</u>

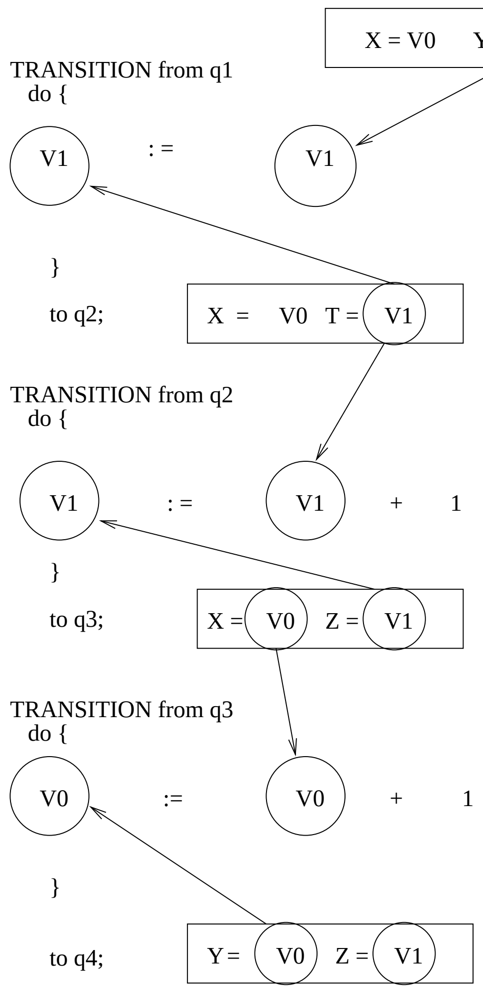


FIG. 5 – *Visualisation du "renommage" des variables.*

5 Conclusion générale de l'apprentissage.

Je me suis principalement intéressé au début de ce travail à comprendre le programme *LIVE*. Ceci a été m'a principale difficulté. En effet, tout en se basant sur un travail déjà fait, il fallait l'améliorer et le compléter en respectant les normes de programmation. Pour cela j'ai dû apprendre la programmation en C^{++}, que je ne connaissais pas, langage du programme *LIVE*. Après l'apprentissage du langage, j'ai étudié *LIVE* au travers d'exemples *IF*, ce qui m'a permis de comprendre l'objectif de mon stage. C'est à dire de connaître les structures employées, pour les utiliser ou les modifier sans trop de contraintes. En effet le programme *LIVE* devait être modifié selon les critères existants et non réécrit totalement. Les objectifs de "renommage" des variables, de la suppression des actions, et la garantie de cohérence ont été atteints. Une extension envisagée serait de considérer tous les types en même temps, car ici je ne considère que des variables de type *INT*. Ensuite il est envisageable de prendre en compte les différents contenus des files d'attente au niveau des états, ainsi leur gestion. Car il faut savoir qu' avec *IF*, nous pouvons augmenter ou diminuer la priorité d'une variable qui se trouve dans une file d'attente, d'où une gestion plus fine de l'utilisation des variables.

6 Bibliographie.

Références

[1] Dorel Marius Bozga *: Vérification symbolique pour les protocoles de communication.* 1999.

7 Annexe.

Le programme *échange*:
Son exécution avec *LIVE*;
Les différentes étapes du "renommage";
Le programme renommé.

Le programme *test*:
Ce programme est présenté pour visualiser les suppressions d'actions.
Son exécution avec *LIVE*;
Les différentes étapes du "renommage";
Le programme renommé.

```
/**********************************************************************/
/*             Visualisation du programme ECHANGE initial.            */
/**********************************************************************/

system echange;

signal
  sign(int);
buffer
  file:queue of sign;

process echange;
var
  x : int;
  y : int;
  z : int;
  t : int;

state
q0:init;
q1;q2;q3;
q4;q5;q6;
```

```
transition

from q0
if(x<0)
do
  output sign(x) to file
to q1;

from q1
do
  t := y
to q2;

from q2
do
   z := t + 1
to q3;

from q3
  do
  y := x + 1
to q4;

from q4
do
  output sign(z) to file
  to q5;

from q5
  do
  x := z+2
to q6;

from q6
do
  output sign(y) to file
  to q0;

/*------------------------------------------------------------------*/
/*                Travail de live et renommage.                     */
/*------------------------------------------------------------------*/

------------------------------
```

```
PROCESS echange
--------------------------
VARS
----
 1 - x
 2 - y
 3 - z
 4 - t
DEAD_vars = {}
STATE q0
-----------
( LIVE_vars = {1, 2, } )
--------------------------
STATE q1
-----------
( LIVE_vars = {1, 2, } )
--------------------------
STATE q2
-----------
( LIVE_vars = {1, 4, } )
--------------------------
STATE q3
-----------
( LIVE_vars = {1, 3, } )
--------------------------
STATE q4
-----------
( LIVE_vars = {2, 3, } )
--------------------------
STATE q5
-----------
( LIVE_vars = {2, 3, } )
--------------------------
STATE q6
-----------
( LIVE_vars = {1, 2, } )
--------------------------
TRANSITION from q0
  if (x < 0)
  do {
    output sign(x) to file
  }
  to q1;
-----------
( DEF = {}; USE = {1, }; AllUses = {1, }
```

```
-------------------------
TRANSITION from q1
  do {
    t:=y
  }
  to q2;
-----------
( DEF = {4, }; USE = {2, }; AllUses = {2, }
-------------------------
TRANSITION from q2
  do {
    z:=(t + 1)
  }
  to q3;
-----------
( DEF = {3, }; USE = {4, }; AllUses = {4, }
-------------------------
TRANSITION from q3
  do {
    y:=(x + 1)
  }
  to q4;
-----------
( DEF = {2, }; USE = {1, }; AllUses = {1, }
-------------------------
TRANSITION from q4
  do {
    output sign(z) to file
  }
  to q5;
-----------
( DEF = {}; USE = {3, }; AllUses = {3, }
-------------------------
TRANSITION from q5
  do {
    x:=(z + 2)
  }
  to q6;
-----------
( DEF = {1, }; USE = {3, }; AllUses = {3, }
-------------------------
TRANSITION from q6
  do {
    output sign(y) to file
  }
```

```
  to q0;
-----------
( DEF = {}; USE = {2, }; AllUses = {2, }
--------------------------
--------------------------
AVG  PROCESS echange    50.00%
WAVG PROCESS echange    50.00%
MAX  PROCESS echange    50.00%


/*******************************************************************/
             Test du renommage
                  maxDeVariable    : 4
                  maxLiveNumber    : 2
                  GetRenownNumber  : 2 + Tmp
/*******************************************************************/



/*-----------------------------------------------------------------*/
/* Version finale que live rend de initialement (sans renommage).   */
/*-----------------------------------------------------------------*/


system echange;

signal
  sign(int);

buffer
  file : queue of
    sign;

process echange;
var
  x : int;
  y : int;
  z : int;
  t : int;
state
q0 :init;
q1;q2;q3;q4;q5;q6;
transition
from q0
  if (x < 0)
```

```
  do {
    output sign(x) to file
  }
  to q1;
from q1
  do {
    t:=y,
    reset y
  }
  to q2;
from q2
  do {
    z:=(t + 1),
    reset t
  }
  to q3;
from q3
  do {
    y:=(x + 1),
    reset x
  }
  to q4;
from q4
  do {
    output sign(z) to file
  }
  to q5;
from q5
  do {
    x:=(z + 2),
    reset z
  }
  to q6;
from q6
  do {
    output sign(y) to file
  }
  to q0;


/*-----------------------------------------------------------------*/
/*              Version avec renommage sur ECHANGE.              */
/*-----------------------------------------------------------------*/
```

```
/* Partie traitement des echanges. */


Possible Exchange
tabRenown[4].&& tabRenown[0]
tabRenown[5].&& tabRenown[1]
Possible Exchange
tabRenown[8].&& tabRenown[4]
Possible Exchange
tabRenown[12].&& tabRenown[8]
Possible Exchange
tabRenown[25].&& tabRenown[21]
Possible Exchange
tabRenown[0].&& tabRenown[24]
tabRenown[1].&& tabRenown[25]


/* Partie initialisation du tableau de renommage. */


V:x TR[0].WR:0  Src:q0  Mq:0 See:1
V:y TR[1].WR:0  Src:q0  Mq:1 See:1
V:z TR[2].WR:-1  Src:q0  Mq:-1 See:1
V:t TR[3].WR:-1  Src:q0  Mq:-1 See:1


V:x TR[4].WR:0  Src:q1  Mq:0 See:1
V:y TR[5].WR:0  Src:q1  Mq:1 See:1
V:z TR[6].WR:-1  Src:q1  Mq:-1 See:0
V:t TR[7].WR:-1  Src:q1  Mq:-1 See:0


V:x TR[8].WR:0  Src:q2  Mq:0 See:1
V:y TR[9].WR:-1  Src:q2  Mq:-1 See:0
V:z TR[10].WR:-1  Src:q2  Mq:-1 See:0
V:t TR[11].WR:0  Src:q2  Mq:1 See:1


V:x TR[12].WR:0  Src:q3  Mq:0 See:1
V:y TR[13].WR:-1  Src:q3  Mq:-1 See:0
V:z TR[14].WR:0  Src:q3  Mq:1 See:1
V:t TR[15].WR:-1  Src:q3  Mq:-1 See:0
```

```
V:x TR[16].WR:-1  Src:q4  Mq:-1 See:0
V:y TR[17].WR:0  Src:q4  Mq:0 See:1
V:z TR[18].WR:0  Src:q4  Mq:1 See:1
V:t TR[19].WR:-1  Src:q4  Mq:-1 See:0


V:x TR[20].WR:-1  Src:q5  Mq:-1 See:0
V:y TR[21].WR:0  Src:q5  Mq:0 See:1
V:z TR[22].WR:0  Src:q5  Mq:1 See:1
V:t TR[23].WR:-1  Src:q5  Mq:-1 See:0


V:x TR[24].WR:0  Src:q6  Mq:1 See:1
V:y TR[25].WR:0  Src:q6  Mq:0 See:1
V:z TR[26].WR:-1  Src:q6  Mq:-1 See:0
V:t TR[27].WR:-1  Src:q6  Mq:-1 See:0
```

```
/*------------------------------------------------------------------*/
/*      Programme ECHANGE renomme final.                            */
/*------------------------------------------------------------------*/


system echange;

signal
  sign(int);

buffer
  file : queue of
    sign;

process echange;
var
  v0 : int;
  v1 : int;
  Tmp : int;
state
q0 :init;
q1;q2;q3;q4;q5;q6;
transition
from q0
  if (v0 < 0)
  do {
```

```
    output sign(v0) to file
  }
  to q1;
from q1
  do {
    v1:=v1
  }
  to q2;
from q2
  do {
    v1:=(v1 + 1)
  }
  to q3;
from q3
  do {
    v0:=(v0 + 1)
  }
  to q4;
from q4
  do {
    output sign(v1) to file
  }
  to q5;
from q5
  do {
    v1:=(v1 + 2)
  }
  to q6;
from q6
  do {
    output sign(v0) to file,
    Tmp:=v0,
    v0:=v1,
    v0:=Tmp
  }
  to q0;


/*******************************************************************/
/*            Visualisation du programme TEST initial.            */
/*******************************************************************/


system test;
```

```
signal
s(int);

buffer
q:queue of s;

process test;
var
a : int;
b : int;
c : int;
d : int;
e : int;

state
idle :init;
fin;q1;q2;f1;f2;f3;f4;f5;f6;

transition

from idle
if (b>0)
do
e:=b
to q1;

from q1
do
a:=b+1
to q2;

from q2
do
b:=e
to idle;

from idle
if (a < c)
do
d:= a+c+a
to f1;

from f1
do
```

```
a:=d
to f2;

from f2
do
a:=d*d*d
to f3;

from f3
do
e:=c+c+c
to f4;

from f4
do
d:=d
to f5;

from f5
do
output s(a) to q
to f6;

from f6
do
a:=a
to fin;



/*-------------------------------------------------------------------*/
/*              Travail de live et renommage sur TEST.               */
/*-------------------------------------------------------------------*/


/*Live analysis for SYSTEM test
------------------------------
PROCESS test
--------------------------
VARS
----
 1 - a
 2 - b
 3 - c
 4 - d
```

```
 5 - e
DEAD_vars = {}
STATE idle
-----------
( LIVE_vars = {1, 2, 3, } )
--------------------------
STATE fin
-----------
( LIVE_vars = {} )
--------------------------
STATE q1
-----------
( LIVE_vars = {2, 3, 5, } )
--------------------------
STATE q2
-----------
( LIVE_vars = {1, 3, 5, } )
--------------------------
STATE f1
-----------
( LIVE_vars = {3, 4, } )
--------------------------
STATE f2
-----------
( LIVE_vars = {3, 4, } )
--------------------------
STATE f3
-----------
( LIVE_vars = {1, 3, 4, } )
--------------------------
STATE f4
-----------
( LIVE_vars = {1, 4, } )
--------------------------
STATE f5
-----------
( LIVE_vars = {1, } )
--------------------------
STATE f6
-----------
( LIVE_vars = {1, } )
--------------------------
TRANSITION from idle
  if (b > 0)
  do {
```

```
    e:=b
  }
  to q1;
-----------
( DEF = {5, }; USE = {2, }; AllUses = {2, }

--------------------------
TRANSITION from q1
  do {
    a:=(b + 1)
  }
  to q2;
-----------
( DEF = {1, }; USE = {2, }; AllUses = {2, }

--------------------------
TRANSITION from q2
  do {
    b:=e
  }
  to idle;
-----------
( DEF = {2, }; USE = {5, }; AllUses = {5, }

--------------------------
TRANSITION from idle
  if (a < c)
  do {
    d:=((a + c) + a)
  }
  to f1;
-----------
( DEF = {4, }; USE = {1, 3, }; AllUses = {1, 3, }

--------------------------
TRANSITION from f1
  do {
    a:=d
  }
  to f2;
-----------
( DEF = {1, }; USE = {4, }; AllUses = {4, }

--------------------------
TRANSITION from f2
```

```
  do {
    a:=((d * d) * d)
  }
  to f3;
-----------
( DEF = {1, }; USE = {4, }; AllUses = {4, }

--------------------------
TRANSITION from f3
  do {
    e:=((c + c) + c)
  }
  to f4;
-----------
( DEF = {5, }; USE = {3, }; AllUses = {3, }

--------------------------
TRANSITION from f4
  do {
    d:=d
  }
  to f5;
-----------
( DEF = {4, }; USE = {4, }; AllUses = {4, }

--------------------------
TRANSITION from f5
  do {
    output s(a) to q
  }
  to f6;
-----------
( DEF = {}; USE = {1, }; AllUses = {1, }

--------------------------
TRANSITION from f6
  do {
    a:=a
  }
  to fin;
-----------
( DEF = {1, }; USE = {1, }; AllUses = {1, }
--------------------------
--------------------------
AVG  PROCESS client     40.00%
```

```
WAVG PROCESS client     40.00%
MAX  PROCESS client     60.00%
--------------------------

/******************************************************************/
             Test du renommage
                  maxDeVariable    : 5
                  maxLiveNumber    : 3
                  GetRenownNumber  : 3 + Tmp
/******************************************************************/



/*----------------------------------------------------------------*/
/* Version finale que live rend initialement (sans renommage).    */
/*----------------------------------------------------------------*/


system test;

signal
  s(int);

buffer
  q : queue of
    s;

process test;
var
  a : int;
  b : int;
  c : int;
  d : int;
  e : int;
state
idle :init;
fin;q1;q2;f1;f2;f3;f4;f5;f6;
transition
from idle
  if (b > 0)
  do {
    e:=b,
    reset a
  }
  to q1;
```

```
from q1
  do {
    a:=(b + 1),
    reset b
  }
  to q2;
from q2
  do {
    b:=e,
    reset e
  }
  to idle;
from idle
  if (a < c)
  do {
    d:=((a + c) + a),
    reset a,
    reset b
  }
  to f1;
from f1
  do {
    a:=d,
    reset a
  }
  to f2;
from f2
  do {
    a:=((d * d) * d)
  }
  to f3;
from f3
  do {
    e:=((c + c) + c),
    reset c,
    reset e
  }
  to f4;
from f4
  do {
    d:=d,
    reset d
  }
  to f5;
from f5
```

```
  do {
    output s(a) to q
  }
  to f6;
from f6
  do {
    a:=a,
    reset a
  }
  to fin;



/*---------------------------------------------------------------------*/
/*                Version avec renommage sur TEST.                     */
/*---------------------------------------------------------------------*/


/* Partie traitement des echanges. */


Possible Exchange
tabRenown[17].&& tabRenown[12]
tabRenown[19].&& tabRenown[14]
Possible Exchange
tabRenown[32].&& tabRenown[27]
tabRenown[33].&& tabRenown[28]
Possible Exchange
tabRenown[35].&& tabRenown[30]
tabRenown[38].&& tabRenown[33]
Possible Exchange
tabRenown[40].&& tabRenown[35]
Possible Exchange
tabRenown[45].&& tabRenown[40]
Possible Exchange


/* Partie initialisation du tableau de renommage. */


V:a TR[0].WR:0  Src:idle  Mq:0 See:1
V:b TR[1].WR:0  Src:idle  Mq:1 See:1
V:c TR[2].WR:0  Src:idle  Mq:2 See:1
V:d TR[3].WR:-1  Src:idle  Mq:-1 See:1
V:e TR[4].WR:-1  Src:idle  Mq:-1 See:1
```

```
V:a TR[5].WR:-1  Src:fin  Mq:-1 See:0
V:b TR[6].WR:-1  Src:fin  Mq:-1 See:0
V:c TR[7].WR:-1  Src:fin  Mq:-1 See:0
V:d TR[8].WR:-1  Src:fin  Mq:-1 See:0
V:e TR[9].WR:-1  Src:fin  Mq:-1 See:0


V:a TR[10].WR:-1  Src:q1  Mq:-1 See:0
V:b TR[11].WR:0  Src:q1  Mq:1 See:1
V:c TR[12].WR:0  Src:q1  Mq:2 See:1
V:d TR[13].WR:-1  Src:q1  Mq:-1 See:0
V:e TR[14].WR:0  Src:q1  Mq:0 See:1


V:a TR[15].WR:0  Src:q2  Mq:1 See:1
V:b TR[16].WR:-1  Src:q2  Mq:-1 See:0
V:c TR[17].WR:0  Src:q2  Mq:2 See:1
V:d TR[18].WR:-1  Src:q2  Mq:-1 See:0
V:e TR[19].WR:0  Src:q2  Mq:0 See:1


V:a TR[20].WR:-1  Src:f1  Mq:-1 See:0
V:b TR[21].WR:-1  Src:f1  Mq:-1 See:0
V:c TR[22].WR:0  Src:f1  Mq:2 See:1
V:d TR[23].WR:0  Src:f1  Mq:0 See:1
V:e TR[24].WR:-1  Src:f1  Mq:-1 See:0


V:a TR[25].WR:-1  Src:f2  Mq:-1 See:0
V:b TR[26].WR:-1  Src:f2  Mq:-1 See:0
V:c TR[27].WR:0  Src:f2  Mq:2 See:1
V:d TR[28].WR:0  Src:f2  Mq:0 See:1
V:e TR[29].WR:-1  Src:f2  Mq:-1 See:0


V:a TR[30].WR:0  Src:f3  Mq:1 See:1
V:b TR[31].WR:-1  Src:f3  Mq:-1 See:0
V:c TR[32].WR:0  Src:f3  Mq:2 See:1
V:d TR[33].WR:0  Src:f3  Mq:0 See:1
V:e TR[34].WR:-1  Src:f3  Mq:-1 See:0


V:a TR[35].WR:0  Src:f4  Mq:1 See:1
```

```
V:b TR[36].WR:-1  Src:f4  Mq:-1 See:0
V:c TR[37].WR:-1  Src:f4  Mq:-1 See:0
V:d TR[38].WR:0  Src:f4  Mq:0 See:1
V:e TR[39].WR:-1  Src:f4  Mq:-1 See:0


V:a TR[40].WR:0  Src:f5  Mq:1 See:1
V:b TR[41].WR:-1  Src:f5  Mq:-1 See:0
V:c TR[42].WR:-1  Src:f5  Mq:-1 See:0
V:d TR[43].WR:-1  Src:f5  Mq:-1 See:0
V:e TR[44].WR:-1  Src:f5  Mq:-1 See:0


V:a TR[45].WR:0  Src:f6  Mq:1 See:1
V:b TR[46].WR:-1  Src:f6  Mq:-1 See:0
V:c TR[47].WR:-1  Src:f6  Mq:-1 See:0
V:d TR[48].WR:-1  Src:f6  Mq:-1 See:0
V:e TR[49].WR:-1  Src:f6  Mq:-1 See:0



/* Notification de quatre actions coupees. */


CUT ACTION
CUT ACTION     /* Cette trace nous indique que nous allons
CUT ACTION        avoir quatre actions en moins. */
CUT ACTION



/*-------------------------------------------------------------------*/
/*       Programme TEST renomme final.                               */
/*-------------------------------------------------------------------*/


system test;

signal
  s(int);

buffer
  q : queue of
    s;
```

```
process test;
var
  v0 : int;
  v1 : int;
  v2 : int;
state
idle :init;
fin;q1;q2;f1;f2;f3;f4;f5;f6;
transition
from idle
  if (v1 > 0)
  do {
    v0:=v1
  }
  to q1;
from q1
  do {
    v1:=(v1 + 1)
  }
  to q2;
from q2
  do {
    v1:=v0
  }
  to idle;
from idle
  if (v0 < v2)
  do {
    v0:=((v0 + v2) + v0)
  }
  to f1;
from f1
  to f2;
from f2
  do {
    v1:=((v0 * v0) * v0)
  }
  to f3;
from f3
  to f4;
from f4
  to f5;
from f5
  do {
```

```
    output s(v1) to q
  }
  to f6;
from f6
  to fin;
```